Projet compilation répartie et délocalisée

Rapport projet Réseau et Système

BEN GAZDALLAH Feten BEN MBARKA Moez
JEMOUI Hanene DOUBLI Lobna

Table des matières

Introduction

La compilation répartie et délocalisée est le but principal de ce projet . Nous devons traiter dans ce contexte l'architecture *Client-Serveur* dans le domaine des réseaux. D'une part nous devons aboutir à des solutions pour gérer convenablement la communication Client-Serveur donc la *délocalisation* et d'autre part bien optimiser les choix en cas de plusieurs serveurs dans le deuxième contexte de *répartition.*

Chapitre 1

Première tranche

1.1 Exigences et analyse des points du sujet

Au niveau de cette première tranche du projet, nous gérons tout simplement la connexion client-serveur et une première étape de la compilation délocalisée. Cette première partie du projet gère principalement les deux fonctionnalités suivantes :

- Double rôle d'un client :
  1. un compilateur C
  2. communication avec le serveur
- Compilation délocalisée :
  cette tâche a bien exploité les avantages du protocole TCP (contrôle de flux...)

1.2 Conception

1.2.1 Gestion des clients

En général un serveur TCP doit traiter plusieurs connexions simultanément. Nous sommes ainsi amenés à penser à la façon avec laquelle gérer les clients. Les différents clients sont gérés par des processus fils chez le serveur (cette solution sera expliquée d'avantage dans la section 1.3 3) .

1.2.2 Gestion des fichiers clients

Un serveur donné peut recevoir plusieurs demandes de compilation avec un même nom du fichier. Ce qui met en relief la nécessité de signer les noms des fichiers auprès du serveur afin que celui-ci puisse différencier les différents fichiers. Rappelons que le serveur accorde un processus fils par client afin de gérer les demandes de ce dernier.

Le pid de ce processus fils sera également utilisé pour signer les fichiers du client localement auprès du serveur.

1.2.3 Protocol de communication

Un protocol de communication entre serveur et client à été proposé par le sujet. Ce protocole se base sur l'échange de messages textes avec l'utilisation de plusieurs mots clés.

Pour gérer ce protocole par le client et le serveur, chacun dispose d'un parseur permettant de parcourir un train de commandes arrivant tout en ignorant les commandes inconnues ou non compris.

1.3 Réalisation et tests

1.3.1 Structure des données

Coté serveur :

Le protocole de communication entre client et serveur impose qu'il n'y a pas d'interaction entre le client et le serveur. Le premier envoie sa requête sous forme d'un train de commandes, dont la fin est signalée par une demi fermeture de la connexion.

A la réception de la requête client, le serveur utilise son parseur pour remplir une structure de données qui lui permettra à la fin de la réception d'exécuter la requête :

```
typedef struct cmd_compilation_serveur_{
  char* cmd; /* La commande */
  int nb_arg ; /* Nombre d'arg */
  char* fichier; /* Le nom du fichier a recevoir */
  char* fichier_tmp ; /* Le nom de fichier temporaire pour le fichier
 généré sur le serveur */
  char* stdout_tmp ; /* Le nom du fichier vers lequel a été redirigé
la sortie standard*/
  char* stderr_tmp ; /* Le nom du fichier vers lequel a été redirigé
la sortie standard d'erreurs*/
  int value ; /*Contient la valeur de retour de la commande de
compilation ou un nom de signal*/

  int taille ; /* La taille du fichier reçu */
  int fdClient; /* Le descripteur du fichier permettant la connexion
                   avec le client demandant la compilation*/
}cmd_compilation_serveur;
```

Coté client :

Le client aussi dispose d'une structure de données permettant de stocker les paramètres de la requête qu'il se prête à faire :

```
typedef struct cmd_compilation_client_{
  char* cmd; /* La commande */
  char* target ; /*optionnel: version du compilateur a utiliser*/
  char* fichier_objet; /* Le nom du fichier objet a recevoir */
```

```
  int value ; /*Contient la valeur de retour de la commande de
compilation ou un nom de signal*/
  int fdServeur; /* Le descripteur du fichier permettant la connexion
    avec le serveur*/
  int repartition; /*1:La répartition est activé, 0 sinon*/
}cmd_compilation_client;
```

1.3.2 Connexion TCP

Coté serveur

La mise en place du serveur est assurée par les modules suivants :

- *init_Serveur.c*
  Ce module contient la fonction *serveur_tcp* qui permet de lancer le serveur sur un port donnée et le met en attente d'une connexion client.
- *demarrage_serveur.c*
  Ce module implémente la fonction principal *démarrer_serveur* qui lance le serveur et boucle sur la fonction d'attente : *attendre_client*.
  A La connexion d'un client, un processus fils est créé pour gérer indépendamment la requête client.

  Ce module ouvre aussi une thread qui se met en écoute sur le port découverte pour gérer les requêtes UDP d'un client(voir 2.2.3).
- *serveur.c*
  C'est le programme principale. Il lance le serveur principal et le serveur découverte.

Coté client

La connexion coté client est assuré par la fonction *connecter* ,du module *client_communication.c*, qui essaie de contacter un serveur sur un port donné. La liste des serveurs potentiels est récupérée du fichier de configuration(voir 2.3.1).

1.3.3 Préparation de la requête de compilation par le client :

Le client est appelé à la ligne de commande avec les différents paramètres de compilation. Le client récupère ses paramètres pour envoyer les commandes correspondantes au serveur.

L'algorithme du choix du serveur à contacter sera détaillé dans le chapitre Deuxième tranche.

1.3.4 Implémentation du protocol de communication :

Les commandes échangées entre un client et le serveur sont orientées lignes commençants par des mots clés définis par le protocol. Le principe d'interprétation de ces commandes se base sur le parcourt du train de commandes arrivant

en capturant les mots séparés par un ou plus d'espace et en essayant d'interpréter chacun de ces mots.

La difficulté principale est le fait que le sens d'un mot capturé dépend du sens du mot précédent ou pour certaines commandes du sens du mot d'avant. Pour gérer ceci, on dispose d'une variable disponible tout le long de la réception d'un train de commande et dont la valeur permet de déterminer le sens de la commande suivante.
Cette variable peut prendre une valeurs parmi plusieurs traduisant les différents mot clés :

```
#define ANY 0 /* On attend n'import quelle cmd */
#define SET 1 /* On a recu SET*/
#define SET_ARGC 2 /* on a recu argc */
#define SET_ARGV 3 /* on a recu argv*/
#define FILE_N 4 /* On recu file et on attend le nom du fichier*/
...
```

Exemple :

Scénario : le client envoie : set argc 3
- Initialement : Last_cmd = ANY

- On lit 'set' : Last_cmd = SET : on attend un nom de variable.
- On lit 'argc' : Last_cmd = SET_ARGC : On attend la valeur de argc.
- On lit '3' : C'est la fin de la ligne. Last_cmd = ANY.

1.3.5 Traitement de la requête client :

Le traitement de la requête d'un client par un processus serveur se fait selon les étapes suivante :
- En parcourant le train de commandes reçu, le serveur remplit une structure *cmd_compilation_serveur* avec les paramètres de la requête.
- En recevant la commande *file*, le serveur génère un nouveau nom de fichier (en concaténant son pid au nom du fichier reçu) et y enregistre le flux suivant cette commande.
- Après la lecture de la totalité du flux reçu, la commande reçue est exécutée tout en redirigeant les sorties standards vers des fichiers temporaires.
- Le serveur envoie la réponse à la requête client : La valeur de retour de la compilation, les sorties standards et éventuellement le fichier objet.

1.3.6 Tests de la première tranche

Exemple d'une compilation délocalisée

Pour lancer la compilation suivante :
`gcc -c -Wall test.c`
Il suffit de lancer :
`crd -c -Wall test.c`
(crd(Compilation Répartie et Délocalisée) : le nom de notre client).

- Précompilation par le client : Le client génère le fichier test.i et l'envoie au serveur.
- Réception du fichier par le serveur : A la réception le serveur renomme le fichier avant de l'enregistrer et lance la commande :
  `gcc -c -Wall test.c`
  Les fichiers générés sur le serveur :

  ```
  8502_test.i : Fichier reçu par le client.
  8502_stdout : La sortie standard de la compilation.
  8502_stderr : La sortie standard d'erreurs de la compilation.
  8502_test.o : Le fichier objet généré.
  ```
- Réception par le client : Le client reçoit les sorties standards de gcc

  ```
  test.c: In function 'main':
  test.c:9: warning: control reaches end of non-void function
  ```

Chapitre 2

Deuxième tranche

2.1 Exigences et analyse des points du sujet

Cette deuxième tranche du projet aborde les points suivants :

- Plus de transparence pour l'utilisateur.
- Gestion des fichiers de configuration : facilitant les tâches du client et du serveur
- Un protocole de découverte simple :

  Les clients échangent dans ce contexte des informations ne nécessitant pas un contrôle de flux particulier ce qui justifie bien le fait que le sujet propose sa gestion en UDP.

2.2 Conception

2.2.1 Plus de transparence pour l'utilisateur

Le premier but de cette tranche est de permettre à l'utilisateur d'utiliser notre client de la même façon qu'un compilateur en local comme gcc.
Ceci sera assuré par la gestion des paramètre de la ligne de commande et les faire passer comme tel à gcc.

Pour faciliter le fonctionnement de nos clients, nous avons pensé à différencier, au niveau de la ligne de la commande, les options propres pour notre client de ceux destinés au compilateur. Ce détail sera précisé dans le chapitre manuel utilisation.

2.2.2 Gestion des fichier de configuration

L'utilisation de fichiers de configuration à été proposé par le sujet pour faciliter l'utilisation du compilateur. Ces fichiers permettent de personnaliser le client et le serveur en modifiant les paramètres généraux : architectures supportées, serveurs disponibles, port découverte...

Le contenu de ces fichiers doit donc être parcouru et analysé avant de traiter les requêtes serveurs et clients.

2.2.3 Choix entre thread et processus pour le protocole de découvert

Concernant la gestion des clients auprès du serveur dans la cadre du protocole de découverte, nous avons fait un certain choix en terme de programmation système. La justification de ce choix doit être précédé d'une certaine précision de la différence entre les threads et les processus.

Threads et processus

Un processus nécessite un changement de contexte étant donné qu'il a son propre espace mémoire ce qui n'est pas le cas des threads dits processus légers qui partagent l'espace mémoire du processus au sein duquel ils sont manipulés.

Ainsi si nous utilisons un processus classique, pour une telle découverte simple, le changement de contexte sera long et presque 90% de ce temps sera consacré à la gestion de mémoire.

Gestion des clients avec un thread

A ce niveau, nous avons opté pour une gestion du client par un thread lors d'une connexion UDP de découverte auprès du serveur.

Nous considérons que la tâche de découverte effectuée à chaque fois par un client auprès du serveur ne nécessite pas vraiment un processus indépendant. Ainsi afin de minimiser les changements de contexte notre processus client aura tout simplement à partager l'espace mémoire du processus serveur.

2.3 Réalisation et tests

2.3.1 Fichiers de configuration

Les fichiers de configuration coté serveur et client utilisent la même syntaxe proposée par le sujet. Pour cette raison, ils utilisent le même module *config.c* pour parcourir les fichiers de configuration.

Structure des données

Pour stocker les informations récupérés des fichiers de configuration, la structure suivante a été utilisé :

```
typedef struct config_file_{
  char** compilers; /* La liste des architectures supportés*/
  int nbCompilers ; /* Nombre d'architecture*/
  char**  gcc_path; /* Les chemins d'accés vers les gcc
```

```
                        correspondants à ces architectures*/
  char* default_compiler; /* L'architecture par défaut */
  int discovery_port; /* Port découverte */
  char** server; /* Tableau des serveurs */
  int nbServers ; /* Le nombre des serveurs */
  int* port; /* Les ports corresponadants */
  char** providers;
  int* etat ; /* tableau indiquant si le serveur
               correspondant est disponible ou pas*/
}config_file ;
```

Le module config.c offre aussi les différents accesseurs pour utiliser cette structure.

Parcourt des fichiers de configuration :

Lors de l'utilisation du fichier de configuration, le client ou le serveur effectue un parsing du fichier pour remplir la structure .
Le parcourt se base sur le même principe que pour l'interprétation des commandes du protocole de communication TCP. Le remplissage de la structure se fait donc en temps réél : un seul parcourt est effectué.

Le tableau *etat* indique la disponibilité des serveurs :
- 1 : Serveur disponible ou n'a pas été contacté.
- 0 : Le serveur n'a pas répondu á une demande de connexion.

Ce tableau utilisé ci-dessous est initialisé avec des 1.

Choix du serveur à contacter par le client :

Avec les fichiers de configuration, le client peut disposer d'un certains nombre de serveurs à contacter. Un algorithme de choix par défaut est utilisé pour faire le choix.
Principe de cet algorithme :

1. On prend le premier le tableau server (C'est aussi le premier en commençant par le début dans le fichier de configuration du client) dont la case correspondante dans le tableau etat est égale à 1.
2. On essaie de contacter ce serveur.
3. Si la connexion s'établit, l'algorithme du choix est terminé.
4. Sinon, on supprime ce serveur : la suppression se fait simplement en mettant 0 dans la case correspondante dans le tableau etat.
5. On retourne à l'étape 1.

Cet algorithme est implémenté par la focntion *choisir_serveur_defaut* du module *test.c*.

A noter aussi, que cet algorithme est appliqué s'il n'y a pas d'option particulière. Par exemple, si on appelle le client avec l'option '–r'(compilation répartie) un autre algorithme est utilisé(Voir chapitre : Troisième tranche).

2.3.2 Protocole de découverte

Ce protocole a été proposé par le sujet pour permettre au client de demander au serveur des informations utiles.

Coté serveur

Au démarrage du serveur, un thread est lancé pour gérer un port découverte. Celui ci doit être désigné dans le fichier de configuration. Dans le cas contraire, le thread est fermé.

Au démarrage, le thread ouvre un socket en mode non connecté UDP et se met à l'écoute sur le port découverte. La lecture d'une commande client est assuré par recvfrom. Cette commande est ensuite passé à la fonction *interpreter_decouverte* implémentée dans le module *serveur_communication.c* qui retourne la réponse du serveur. Dans le cas où la commande reçue n'est pas comprise le message “Commande découverte non supportée.” est retourné. Ensuite, cette réponse est envoyée à l'aide de sendto. Les coordonnées du récepteur sont récupérés dans une variable de type sockaddr remplie par recvfrom à la réception.

2.3.3 Coté client

Le client peut être appelé à la ligne de commande pour envoyer une commande découverte(Plus de détails dans le manuel utilisation).

La commande est envoyée en diffusion sur le port de découverte indiqué dans le fichier de configuration.
Pour ne pas bloquer longtemps sur la commande recvfrom si aucune réponse n'est pas reçu, une alarme est déclenchée puis désactivée à la réception d'une réponse.

2.3.4 Tests de la deuxième tranche

makefile

Après cette tranche, notre client peut être utilisé à la place de gcc dans un makefile. Pour faire des tests nous l'avons utilisé pour compiler nos codes sources :

```
CC=./crd
LD=gcc
CCFLAGS= -Wall
LDFLAGS= -lsocket -lnsl -lrt
```

Commande découverte

Une commande découverte peut être lancée de la manière suivante :

```
./client --discovery compilers
```

La réponse sera de la forme :

```
Réponse du serveur :scorpion: à la commande :compilers:
0.0.0.0 1037 provides v0
```

Chapitre 3

Troisième tranche

3.1 Exigences et analyse des points du sujet

- Compilation délocalisée mais également répartie
- Bonne gestion de la répartition des clients selon les serveurs disponibles

3.2 Conception

3.2.1 Communication des clients

Avant de détailler notre solution conçue, précisons que nous avions prévu que la gestion des clients doit se faire selon deux niveaux :

1. les clients locaux sur une machine (plusieurs clients sur tanit)
2. les clients sur autres machines (un client sur tanit et un autre sur big)

Fichier de communication des clients locaux

Nos clients consultent et modifient un fichier décrivant l'état des différénts serveurs occupés en précisant à chaque fois le pid du client , le nom du serveur et la taille du fichier à compiler.
Deux solutions se présentent :

1. un seul fichier de communication :
   décrivant tous les serveurs utilisés et la taille des fichiers manipulés.
2. un fichier par serveur :
   donnant la taille totale des fichiers manipulés par le serveur et le nombre des clients en question.

Ces deux solutions seront plus détaillées ci-dessous en précisant les avantages et les inconvénients de chacune dans le cadre de notre application, ce qui justifiera à priori notre choix final.

1. Fichier unique de communication :
   Une première idée est de faciliter la communication des clients locaux par

un fichier unique ayant la format suivante.

pid_client	nom_serveur	taille_fichier

TAB. 3.1 – champs du fichier de communication

ainsi le client sera supposé faire un parcours de ce fichier et choisir éventuellement le meilleur serveur c'est à dire le moins utilisé mais également manipulant la plus petite taille des fichiers à compiler. Le problème d'une telle solution est le cout de parcours et de mise à jour du fichier car à la fin de compilation du fichier, le client doit chercher la ligne contenant son pid et l'effacer pour libérer ainsi le serveur en question.
Ainsi nous étions amenés à concevoir une solution moins couteuse en utilisant un fichier par serveur.

2. Fichier par serveur :
   Nous rappelons qu'à ce niveau nos solutions de fichiers décrivant l'état des serveurs sont vraiment locales c'est à dire pour des clients locaux . Comme la solution précédente est coûteuse, une deuxième solution est d'avoir un fichier pour chaque serveur appelé *nomserveur.log* contenant une seule ligne de cette forme : taille nbutilisé Si un client donné est le premier à utiliser ce serveur alors il crée ce fichier local et l'initalise, quand la compilation est terminée ou mal déroulée le client diminue la taille selon la taille de son fichier et décrémente le nombre de fois utilisées pour ce serveur.

Gestion de la coordination avec les clients non locaux

Idée : avoir au niveau du serveur d'un tableau consultable via le protocole de découverte.
Ce tableau contient le nombre de requetes selon les adresses IP .

IP	NbreRequetes
A.B.C.D	6
E.F.G.H	3
I.J.K.L	4

TAB. 3.2 – Tableau d´état auprès du serveur

Ainsi un client donné ayant déjà contrôlé l'utilisation des clients locaux, ne contrôle pas la case du tableau ayant sa même adresse IP et controle les autres pour avoir une idée sur le nombre de requetes envoyées depuis autres clients.
Par exemple selon notre tableau, si le client correspond à l'adresse IP A.B.C.D, alors il va considérer que le serveur a un nombre de requêtes égale à 7.
Pour le moment, cette solution peut être lente mais également pose un problème de coordination puisque un serveur donné peut ne pas être trop utilisé par les clients locaux mais être utilisé ailleurs, ainsi notre client peut croire avoir fait un bon choix de serveur mais découvrir le contraire suite à une connexion de découvrte !

Gestion du partage du fichier de communication et gestion des sémaphores

Chaque processus client utilise le même algorithme d'utilisation de la sémaphore du fichier de communication.
Le principe est le suivant :
Le processus client verrouille avec la sémaphore, il entre en section critique lors de la consultation des fichiers d'état des serveurs , afin de garantir la validité des informations qu'il est entrain de lire.
Ensuite, il modifie le fichier d'état du serveur choisi en ajoutant la taille de son fichier et en incrémentatnt le nombre de clients de ce serveur.
A ce moment, il quitte la section critique en libérant la sémaphore et il envoie la commande de compilation au serveur choisi.
Après avoir effectué la compilation délocalisée ou ne pas avoir pu se connecter au serveur , notre client réutilise la sémaphore pour réaccéder au fichier du serveur en question et le mettre à jour.
L'algorithme est le suivant :

```
Algorithme GestionDesFichiersDesServeurs
Variable
    sémaphore sem
    { cette sémaphore permet de gérer les }
    { accès conccurents aux fichiers des serveurs }
Début
    Verrouiller(sem)
    LectureFichiersDesServeurs
    choisirServeur
    { cet algorithme sera détaillé dans la section 3.2.2 page 15 }
    EcritureFichierDuserveur
    Deverrouiller(sem)
    EnvoiCommandeCompilation
    Verrouiller(sem)
    mettreAjourFichierServeurUtilisé
    Deverrouiller(sem)
Fin
```

3.2.2 Algorithme de sélection de serveur

Nous rappeleons que l'algorithme de choix du meilleur serveur est géré localement pour le moment, c'est à dire nous permettons qu'aux processus sur la même machine de l'appliquer.
Nous rappelons également que chaque serveur possède un fichier `nomserveur.log` décrivant son état. L'algorithme de sélection de serveur considère trois paramètres :

1. le nombre des clients
2. la taille des fichiers
3. le nombre des serveurs disponibles

Mise en place de l'algorithme

```
Algorithme choisirServeur
Variable
    entier nb
    { nb est le numéro du serveur auprés du client }
Début
    Si  (repartitionSouhaitée) Alors
        { si l'utilisateur souhaite une compilation répartie }
        nb ← selectionnerServeur
        { voir l'algorithme suivant }
    Sinon
        nb ← choisirServeurDefaut
        { le numero du premier serveur dans le fichier de }
        {  configuration non déja utilisé }
    Si nb=-1 Alors
        afficher(Aucun serveur ne peut être contacté)
    Si nonConnectionServeur(nb) Alors
        supprimerServeur(nb)
        choisirServeur
Fin
```

L'algorithme précédent de choix final du serveur est basé sur l'algorithme de sélection de serveur suite à la consultation des fichiers des serveurs.
L'algorithme est le suivant :

```
Algorithme selectionnerServeur
Variable
    entier nbserveur,n,id,t,index
    { nbserveur est le nombre des serveurs disponibles }
nbserveur←()nbserveursdisponibles()
{ retourne le nombre des serveurs disponibles }
{ selon le client }
Si nbserveur=0 Alors
    retourner -1
n←()nbutilisation(premierserveur)
{ nombre d'utilisation du premier serveur }
Si nbserveur=1 || n=0 Alors
    retourner id
    { si on a un seul serveur on le sélectionne par défaut }
    { ou si le premier serveur n'est pas occupé }
Pour i variantDe 1 à nbserveur-1 Faire
    n←()nbutilisation(id)
    Si n=0 Alors
        retourner id
    { retourne le premier disponible n'ayant aucune tâche à faire }
    { on arrête la recherche d'un bon serveur }
    Si n<5 Alors
        { si le serveur est utilisé moins que 5 fois }
        t recoit()tailledesfichier(id)
        Si min > t Alors
            min ←() t
            index ←() i
        { chercher le moins chargé par rapport }
        {  aux tailles des fichiers qu'il s'en occupe }
    { sinon on retourne le premier sur la liste par défaut }
 retourner index
```

3.3 Réalisations et tests

3.3.1 Structure des données

Pour stocker les informations recupérées des fichiers des états des serveurs, la structure suivante est utilisée :

```
typedef struct serveurs_utilises_{
  char** serveurs;
  int* hashage ;
```

```
  int* taille ;
  int* nbUtilisations ;
  int nbServeurs ;
}serveurs_utilises;
```

Le module etat_serveurs.c offre les accesseurs permettant l'utilisation de cette structure.

3.3.2 Lecture des fichiers des états des serveurs

Ces fichiers sont le moyen de communication entre les processus, donc ils doivent être lus avant de lancer l'algorithme de choix du serveur. Les noms de serveurs sont recupérés du tableau *serveurs* de la structure config_file traduisant le contenu du fichier de configuration.

Cependant on vérifie avant de prendre en compte un serveur que son état est encore à 1 (plus de détails pour le tableau etat dans le chapitre Deuxième tranche). De cette façon , la liste des serveurs enregistrés dans la structure serveurs_utilises est un sous ensemble de la liste complète des serveurs. Pour garder l'index du serveur dans le tableau de config_file, on garde un tableau *hashage* donnant cet index pour tout serveur de la structure serveurs_utilises.

D'autre part, on vérifie l'existence des fichiers des états des serveurs, dans le cas contraire, ceux-ci sont créés et initialisés.

3.3.3 Tests de la troisième tranche

Pour pouvoir utiliser la répartition, il faut configurer le makefile comme indiqué dans la partie manuel utilisation.

Exemple

Compilation de trois fichiers avec la répartition(voir le dossier test fourni). On dispose de trois serveurs ouverts.

Résultat obtenues

La lecture du fichier log du client indique que chacun des fichier a été envoyé à un serveur différent :

```
...

3636--->Thu May 18 23:49:09 200: J'ai choisi : 0:balance.

3636--->Thu May 18 23:49:09 200: Je cherche l'IP de balance...


3636--->Thu May 18 23:49:09 200: J'ai contacté balance.

...
```

```
3637--->Thu May 18 23:49:09 200: J'ai choisi : 1:germinal.

3637--->Thu May 18 23:49:09 200: Je cherche l'IP de germinal...


3637--->Thu May 18 23:49:09 200: J'ai contacté germinal.

...

3638--->Thu May 18 23:49:10 200: J'ai choisi : 2:hamilcar.

3638--->Thu May 18 23:49:10 200: Je cherche l'IP de hamilcar...


3638--->Thu May 18 23:49:10 200: J'ai contacté hamilcar.
...
```

3.4 Problèmes rencontrés

3.4.1 Gestion des sémaphores

Le problème majeure rencontré est lié au blocage de la sémaphore suite à une interruption ou un signal :
Si notre processus *client* reçoit un signal il meurt sans déverrouiller la sémaphore . Ainsi le prochain processus client ne pourra pas l'utiliser.
Même si nous avons pensé à un certain moment à avoir recours aux masquage des interruptions mais celà parait un peu incohérent avec le fait que les sémaphores gère le masquage des interruptions au niveau système.
Par défaut de temps, nous n'avons pas réglé ce problème.

Chapitre 4

Description globale et maintenance

4.1 Schéma global des modules

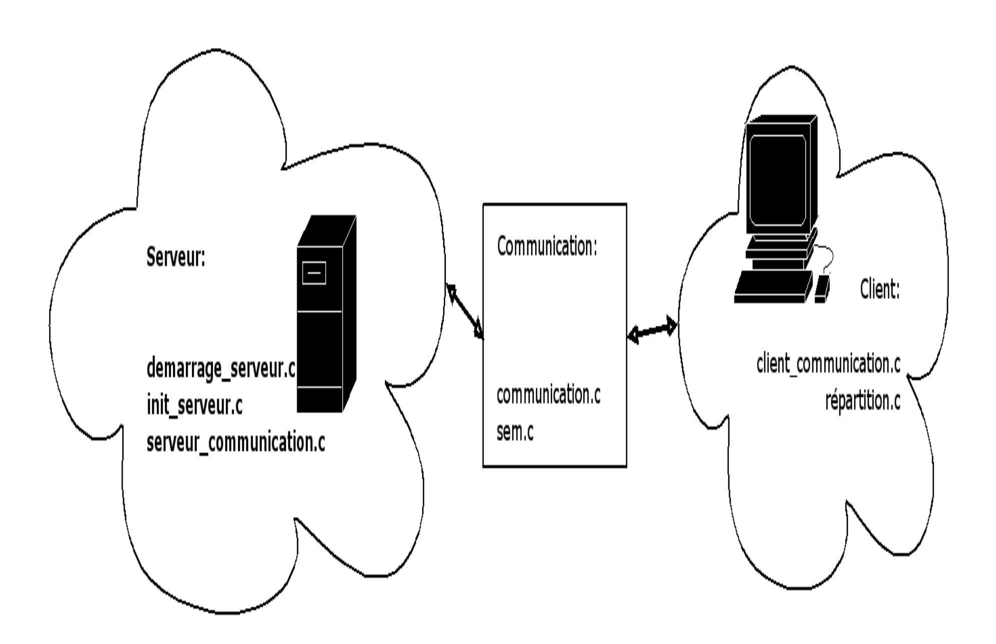


FIG. 4.1 – Vue globale

Le schéma 4.1 précise l'organisation des modules principaux selon les fonctionnalités (Client, Serveur ou communication).
Ces modules et leurs fonctions seront détaillés d'avantage dans la section 4.2 suivante .

4.2 Description des modules

Nous détaillons dans cette partie le code des modules principaux :
c'est à dire que les fonctions principales (décrivant nos algorithmes et nos choix de base) et leurs utilité.

4.2.1 Serveur

Dans le répertoire *server* de notre projet,nous avons le code du module *Serveur*.
Rappelons que les fonctionnalités principales de notre serveur sont les suivantes :

init_serveur

Ce fichier *init_seveur.c* contient les fonctions de base pour initialiser le serveur pour les connexions.

- La fonction *serveur_tcp* :

  `int serveur_tcp(int numero_port)`
  Cette fonction permet de démarrer un service TCP sur le port indiqué en :

  1. Créant une socket (*socket()*)
  2. initialisant la socket (*bind()*)
  3. ouverture du service TCP (*listen()*)
- La fonction *attendre_client* pour l'attente d'un client :

  `int attendre_client(int fdServeur)`

demarrage_serveur

Ce fichier *demarrage_serveur.c*
permet au serveur de traiter les processus clients selon les différents contextes :

1. Compilation (avec processus)
2. Découverte (avec threads)

- La fonction *demarrer_serveur* :

  `int demarrer_serveur(int numero_port, char fichier[])`
  Cette fonction gère les points suivants :

  1. fonctionnemet du serveur (boucle infinie)
  2. traiter les commandes des clients
  3. traiter les signaux des processus clients (fils) et serveur (père)
- La fonction *demarrer_decouverte* :

```
void* demarrer_decouverte(void* t)
```
Elle crée un thread qui se charge de recevoir les requêtes UDP de découverte et d'y répondre.
- La fonction *demarrer_serveur* :

```
void demarrer_serveur_decouverte()
```
Cette fonction crée tout simplement le thread nécessaire pour la fonction précédente.

- Les focntions de traitement des signaux :

  1. La fonction *fin_serveur* :

     ```
     void fin_serveur()
     ```
     pour traiter les signaux d'arrêt du serveur.
  2. La fonction *fin_fils* :

     ```
     void fin_fils()
     ```
     traitant les signaux des processus clients fils.

serveur_communication

Ce fichier traite explicitement les requêtes des clients à l'aide de fonctions suivantes :

- La fonction *traiter_commandes_client* est la fonction principale de ce module (utilisant les fonctions suivantes) :

```
void traiter_commandes_client(int fd)
```
Elle traite la commande client avec *fd* le descripteur de fichier sur lequel la commande du client a été récupérée.
- La fonction *envoyer_reponse_serveur* :

```
void envoyer_reponse_serveur()
```
Envoie le résultat de la compilation(sorties et fichier objet) au client.
- La fonction *executer_commande_client* :

```
void executer_commande_client()
```
redirige les sorties standard vers des fichiers temporaires (pour stocker les messages d'erreur) puis exécute la commande client.
- La fonction *interpreter* :

```
int interpreter(char* cmd)
```
Elle interprète les mots clés de la commande client.
- La fonction *liberer_cmd_server* :

```
void liberer_cmd_server()
```

Libèrant la structure *cmd_server* utilisé.Elle est à appeller à la fin de l'envoie de la réponse du serveur.
- La fonction *interpreter_decouverte* :

  `char* interpreter_decouverte(char* buf)`
  interprète une commande reçu sur le port découverte

4.2.2 Client

client.c

C'est le fichier principal de traitement du client.

- La fonction *choisir_serveur_defaut* :

  `int choisir_serveur_defaut()`
  Cette fonction implémente l'algorithme de choix par défaut d'un serveur.Ça revient à choisir le premier serveur dans la structure *config_f* de notre client.
- La fonction *init_connexion* :

  `int init_connexion(int indexServeur)`
  Initialisant tout sinplement la connexion au serveur indiqué par son index *indexServeur*.
- La fonction *traiter_arguments* :

  `int traiter_arguments(int argc, char** argv)`
  Cette fonction traite les arguments de la commande de compilation.

client_communications.c

Au niveau de ce fichier , nous avons les commandes directement liée à la communication avec le serveur, l'envoi et la réception des réponses qui en découlent, soit pour une découverte soit pour une
- La fonction *connecter* :

  `int connecter(char* server, int port)`
  Le client se connecte au serveur de nom *server* via le port numéro *port*.
- La fonction *envoyer_commande_compilation* :

  `void envoyer_commande_compilation(int argc, char** argv, int fd)`
  Permettant à notre client d'envoyer une demande de compilation au serveur.
- La fonction *envoyer_commande_decouverte* :

  `int envoyer_commande_decouverte(char* commande)`
  Permettant à notre client d'envoyer une demande de compilation au serveur.

repartition.c

- La fonction *choix_serveur* :

  ```
  int choix_serveur(char* fichier)
  ```
  Récupère le maximum des informations à partir des fichiers log des serveurs et exécute l'algorithme de choix (voir chapitre de la troisième tranche) et retourne l'index du serveur à utiliser, avec *fichier* est le nom du fichier à compiler.
  Cette fonction fait appel à la fonction suivante de sélection du serveur.
- La fonction *decider* :

  ```
  int decider()
  ```
  Elle décide quel serveur utilisé selon les informations récupérées à partir des logs des serveurs.

4.2.3 Communication

Ce module concerne les fonctionnalités globales de communication pour les clients et les serveurs.

communication.c

- Communication en UDP
  1. La fonction *recevoir_UDP* :

     ```
     int recevoir_UDP(int sc,  struct sockaddr_in* sc_from, char* dest)
     ```
  2. La fonction *envoyer_UDP* :

     ```
     int envoyer_UDP(int sc, struct sockaddr_in* sc_addr, char* buff)
     ```
- Communication en TCP
  1. La fonction *recevoir_fichier* :

     ```
     int recevoir_fichier(int fd, int taille, char* dest)
     ```
  2. La fonction *envoyer_fichier* :

     ```
     int envoyer_fichier(char* nom, int sc_fd)
     ```

sem.c

Nous avons utilisé ce fichier dans la troisième tranche pour gérer la communication locale des clients.
Ainsi ce fichier contient les primitifs d'utilisation des sémaphores. Les fonctions sont les suivantes :

```
sem_t* initialiser_sem(int etat)
```

Cette fonction initialise la sémaphore soit avec la valeur 1 (déverrouillée) ou 0 (pour déverrouiller).

```
void fermer_sem()
sem_t* verrouiller_sem(sem_t* sm)
void deverrouiller_sem(sem_t* sm)
```

Chapitre 5

Evolution et conception avancée

5.1 Utilisation des threads pour la compilation

Lors de la conception de la troisième tranche, nous avons abordé dans la section 2.2.3 page 8 la possibilté d'optimisation d'utilisation de mémoire possible avec les threads.
D'autre part, dans la première tranche (voir la section 1.2.1 page 2) les clients sont considérés via des processus fils auprès du serveur.
Une conception plus avancée aurait pu permettre d'optimiser encore plus ce point en considérant les clients comme des threads auprès du serveur dans tous les contextes : Compilation ou découverte.

5.2 Gestion des clients non locaux

Comme nous l'avons déjà précisé dans la section 3.2.1 13, pour notre algorithme actuelle de la répartition de charge selon les états des serveurs(occupation et taille des fichiers manipulés) nous n'avons pas implémenté l'algorithme d'optimisation des choix en fonction des clients distants par rapport à notre client tentant de faire le bon choix(sur des autres machines).

5.2.1 Extension du protocole de découverte

Plusieurs commandes découvertes peuvent être très intéressantes pour l'utilisateur, comme ceux proposés par le sujet dans la tranche 4, pour connaître l'occupation d'un serveur. Ce type de commande peut être utiliser dans l'algorithme de choix pour prendre en compte les clients d'autres machines.

Chapitre 6

Manuel d'utilisation

6.1 Client

6.1.1 Fichier de configuration

Le fichier de configuration pour le client est : client.conf, il doit à être dans le même dossier que l'exécutable.

6.1.2 Utilisation

Le programme client peut prendre un nombre illimité de paramètre á la ligne de commandes. Les paramètres propres à notre programme sont distingués (de ceux destinés à gcc) par le faite qu'ils sont précédés par '–' et doivent toujours précédés les paramètres de gcc

Les paramètres supportés :
- '–discovery' suivie de la commande découverte : envoie une commande découverte.
- '–v' suivi d'un nom d'architecture : Indique au client d'utiliser le compilateur de cette architecture.
- '–r' : Indique au compilateur d'utiliser l'algorithme de répartition pour le choix du serveur.

A priori, les principaux paramètres de gcc, sauf ceux induisant une redirection de flux, peuvent être utilisés. Les paramètres comme '-o' de gcc, sont passés comme tel au serveur, cependant celui-ci ne sait pas recupérer la destination du fichier objet. Le nom de celui ci, est toujours le nom du fichier .i reçu mais avec l'extension .o.

Une autre contrainte imposée par la manière de parcourir les paramètres : Le nom du fichier à compiler doit être le dernier paramètre.

6.1.3 Exemples

- Commande découverte :
  `./client --discovery compilers`
- Fixer l'architecture :

```
./client --v sunOs -c test.c
```

- Répartion :
  Pour utiliser la répartition, il faut configurer le makefile :
  ```
  CC=./client --r
  ...
  ```
  puis :
  `make -j`

  Un fichier makefile.crd est fourni avec le code dans le dossier src/client qui permet de compiler notre projet avec l'option de répartition.

6.2 serveur

6.2.1 Fichier de configuration

Le fichier de configuration pour le client est : client.conf, il doit à être dans le même dossier que l'exécutable.

6.2.2 Utilisation

La configuration du serveur se fait principalement avec le fichier de configuration. Au lancement il prend à la ligne de commande le numéro de port principal.

6.2.3 Exemple

```
./serveur 1037
```

6.3 Fichiers de log

Le client et le serveur génère chacun un fichier de log : crd.log. Ce fichier est créé dans le même dossier que l'exécutable. Il offre la possibilité de voir les principales opérations effectuées par le serveur ou le client.
Les lignes du fichier de log ont un format spécifiques :

```
3636--->Thu May 18 23:49:09 200: Nouvelle session...
pid     date et heure            message
```

Conclusion

Ce projet combinant la programmation réseau et système était une bonne occasion pour apprécier la mise en place de nouveaux protocoles nécessaires pour nos applications informatiques qui sont basées sur les protocoles classiques (TCP, UDP...).
Le contexte de ce projet était riche dans la mesure où nous étions amenés à trouver des algorithmes non seulement efficaces mais surtout optimaux pour des raisons techniques ou afin d'améliorer nos conceptions.
Certains points de ce projet peuvent encore être des points de développement mais qui n'ont pas éte traités par défaut de temps.

Table des figures